AD-A066 442   STANFORD UNIV CALIF SYSTEMS OPTIMIZATION LAB   F/G 12/1
MODULES TO AID THE IMPLEMENTATION OF LP ALGORITHMS.(U)
DEC 78   L NAZARETH   N00014-75-C-0267
UNCLASSIFIED   SOL-78-28   NL

LEVEL II

Systems
Optimization
Laboratory

Department of Operations Research
Stanford University
Stanford, CA 94305

79 03 23 021

LEVEL II

SYSTEMS OPTIMIZATION LABORATORY
DEPARTMENT OF OPERATIONS RESEARCH
Stanford University
Stanford, California
94305

AD A066442

MODULES TO AID THE IMPLEMENTATION OF LP ALGORITHMS.

by

L. Nazareth

TECHNICAL REPORT SOL-78-28
December 1978

Dec 78

23p.

DDC
RECEIVED
MAR 27 1979

408765

Abstract

Implementing large scale LP algorithms is a difficult, laborious and often poorly rewarded task. This is particularly true of algorithms which exploit the structure of the LP matrix. For this reason many algorithms have been proposed in the literature, but few have been turned into good computer codes. Very little is known about the relative performance of different algorithms. In this paper we discuss some of the suggestions that have been made for alleviating this problem and describe an approach based upon a carefully defined collection of subroutines which are designed to aid the task of implementing and comparing LP algorithms. These subroutines or modules may be regarded as the 'primitives' of a language for implementing experimental LP algorithms, particularly algorithms which exploit matrix structure. A set of such modules is described. These have been implemented in FORTRAN, and user documentation is available.

Acknowledgment

MODULES TO AID THE IMPLEMENTATION OF LP ALGORITHMS

by

L. Nazareth

1. Introduction

Implementing large scale LP algorithms is a difficult, laborious and often poorly rewarded task. This is particularly true of algorithms which exploit the structure of the LP matrix. For this reason many algorithms have been proposed in the literature, but few have been turned into good computer codes. Very little is known about the relative performance of different algorithms.

In this paper we discuss some of the suggestions that have been made for alleviating this problem, and describe an approach based upon a carefully defined collection of subroutines which are designed to aid the task of implementing and comparing LP algorithms. These subroutines or modules may be regarded as the 'primitives' of a language for implementing experimental LP algorithms, particularly algorithms which exploit matrix structure. A set of such modules is described. These have been implemented in FORTRAN, and user documentation is provided. (We developed these modules as a first stage in the implementation of a nested decomposition code for staircase structures, but they were purposely designed for use in other contexts, and they may also be a useful educational aid.)

This has been primarily a software organization and development effort, and we have drawn upon the work of many different researchers in the field. We do not claim that the modules are quality software, but we hope that they

1

will evolve in the direction of quality software, e.g., through much more exhaustive testing of the modules.

In the concluding section, we discuss future directions and the benefits to be gained from an effort such as this one.

2. Background

We have discussed the overall framework for optimization software development elsewhere, see Nazareth [1], [2]. In summary, we distinguish between implementations designed primarily to aid algorithms or code development and implementations which are intended for production runs and are thus designed primarily to solve user problems. The former are called algorithm/code oriented implementations (or sometimes experimental implementations) and the later user/problem oriented implementations. Each of these can be further tailored to a particular compiler and machine configuration, resulting in different program realizations (see Boyle and Dritz [3]).

In Nazareth [1], three different approaches to algorithm/code oriented software were also discussed. In the first approach one seeks to develop a suitable high level language. This language permits highly readable programs to be written with relative ease in the vernacular of applied mathematics, and serves as a medium for communicating algorithmic ideas precisely (MPL, see Dantzig et al. [4], is an example of such a language. Another example is the Speakeasy language of Cohen et al. [5]. This latter language also provides a mechanism which enables a user to extend the language to suit his individual needs). In the early creative stages of algorithm development, such a language is very useful, since new ideas can be quickly implemented

and tested out, and the computational experience obtained often results in new insights and developments. Thus the major features of an algorithm can be laid out. Note however that a 'quick and dirty' implementation can usually be run only on toy problems, and that procedures which are numerically sound and efficient in terms of time and storage, are difficult to write in any language. Other pros and cons of this first approach are discussed in Nazareth [1].

In the second approach, one attempts to introduce some limited flexibility into software designed primarily to solve user problems, by making some of the high level components available to the algorithm developer. For example, in the MPSX/370 system, a number of high level routines, listed in Figure 1, can be accessed through a control language and used to construct algorithms. Another example is the SEXOP system, Madsen [6], see Fig. 2, in which the major components are geared toward problem manipulation. Note that in both cases one is greatly constrained by the overall system, particularly by its data structures.

The third approach complements the first, by placing much more emphasis on aids for developing a numerically sound experimental implementation, once some of the major features of an algorithm have been laid out (for example, using an implementation in MPL). In this third approach, one seeks to identify the components that are used to build an LP algorithm, to specify them cleanly and carefully, and to implement them in a manner which makes them flexible and easy to use. These modules can also be thought of as the 'primitives' or 'basic operators' of a language for building LP algorithms. Since our aim is to produce software which is immediately

3

FIGURE 1: ALGORITHMIC PROCEDURES IN MPSX|370

SETLIST (produces internal translation of variable)   FTRANL1

INVALUE (match a list of names)   FTRANU1 ⎫
                                                                  ⎬ (Forward and backward
GETVEC1 (moves column)   BTRANL1 │   transformations)
                                                                  ⎭
FIXVEC1 (computes basics)   BTRANU1

POSTMUL   CHUZR1   (choose row)
         (matrix-vector operations)
PREMUL

PRICEP1 (pricing)   INVCTL1 (should inversion be performed)

4

FIGURE 2: MAJOR COMPONENTS OF SEXOP


```
                SETUP     (read in LP)
                ADDCOL    (add new column )
                REPCOL    (replace column )                    DATA
                SETCOL    (examine column )                    AREA
USER            XCOST     (change objective function )
                XRHS      (change rhs)
                LINEAR    (involve solver)
                PERTURB   (parametric analysis of rhs)         BLACK
                                                               BOX
                TRADOF    (parametric analysis of objective)
```
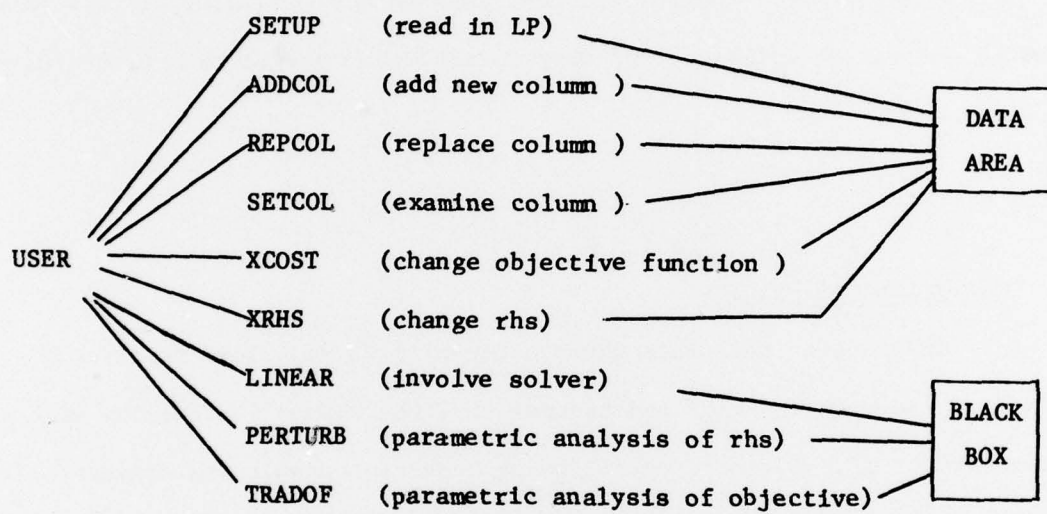
and widely usable, we think that they should be implemented in FORTRAN (they could equally well be implemented in another high level language and our effort can be regarded as another contribution to identifying what should go into a language for optimization; see also Land and Powell [7], Dantzig et al. [8]). Experimental codes which are constructed from these modules should be executable on real life problems, not just on toy data; thus attention should be paid both to efficient data representations and to robustness. Finally the modules should serve as an educational aid.

A set of such modules are described in the remainder of this paper. How the set can be enlarged, and ways to make them easier to use, are discussed in Section 5.

3. Description of Modules

The modules fall into three categories as described below. The following overview is brief and assumes that the reader is familiar with the theory of LP. For more detail on each module consult the documentation, and for a surfeit of detail check the listings (see Section 4).

3.1. Problem Oriented Modules

Problem oriented modules serve as an 'interface' between the user's LP problem and the routines which solve it.

LP problems are usually specified on a tape in standard MPS input format (see [9] and Figure 4) while LP routines usually work on packed matrix representations. A set of modules have been provided to aid in the reading of an LP tape and setting up of the data structures.

6

Typical structured LP matrices are illustrated in Fig. 3.

Block Angular	Dual Block Angular	Staircase

A_1 A_2 A_3 A_4 B_1 A_1 A_1

B_1 B_2 A_2 B_1 A_2

    B_2 B_3 A_3 B_2 A_3

        B_3 B_4 A_4 B_3 A_4
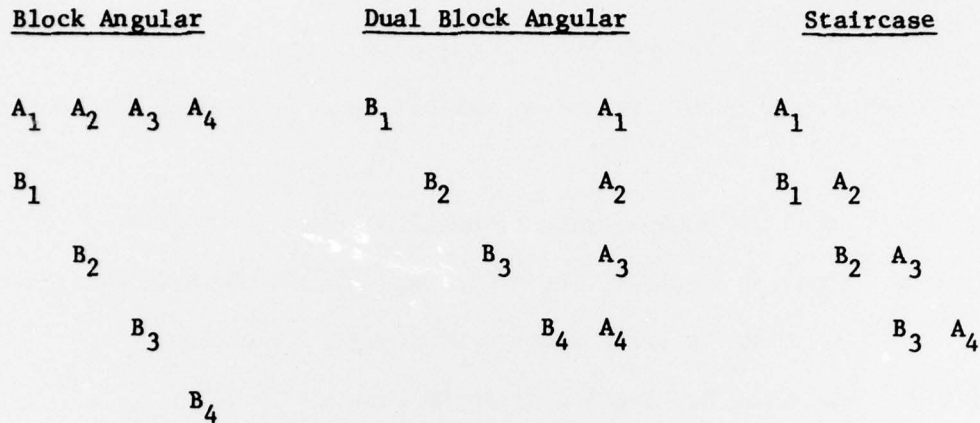
            B_4

FIGURE 3

An LP routine designed to take advantage of structure may require a particular representation of data. Therefore it would be inappropriate to provide a general input routine. Instead we provide components from which a suitable input routine can be built. These modules can be used in conjunction with those of Section 3.2.1, to set up the data representation as required. For example, given a block angular LP matrix, it may be necessary to set up each

$$\begin{pmatrix} A_1 \\ B_1 \end{pmatrix}$$

as an LP matrix with consecutively numbered rows. A call to module PREADC (see below) will set up a packed matrix with rows numbered according to the original matrix. Following this by a call to ADRNDX (see Section 3.2.1) will renumber the rows to be consecutive.

7

The problem oriented modules in this version of the collection are as follows: (They were adapted in major part from the input routines of MINOS, Saunders [10]. However, since we segmented the routine and altered several portions of the code to suit our needs, responsibility for errors rests with us, and should in no way reflect upon the source of the code.)

o PREADR   Read the ROWS section of an LP matrix

o PREADC   Given a specified set of columns in the COLUMNS section, build a column list/row index packed data structure.

o PRDRHS   Read the RHS section of an LP matrix

o PREADB   Read in the BOUNDS section of an LP matrix (see also Section 3.2.2).

o PCHKST   Perform checks and gather statistics about the matrix.

The manner in which a matrix is packed is illustrated in Fig. 3 for a specific example. By writing an input routine which calls the problem oriented modules the user can translate the MPS input into a packed data representation, suitable for his LP algorithm.

3.2. Algorithm Oriented Modules

These are the basic building blocks from which algorithms are constructed. They fall into two categories.

8

FIGURE 4

LP

$$\min \quad x_1 + x_2 + x_3$$

$$\text{s.t.} \quad 2x_1 + \quad 3x_3 \leq 10$$

$$4x_2 + 5x_3 \leq 20$$

$$100 \geq x_1 \geq 0, \quad x_2 \geq 0$$

TABLEAU

Row names	CLM1	CLM2	CLM3		RTH
OBJ	1.	1.	1.		0
RWN1	2.	0	3.	$\leq$	10.
RWN2	0	4.	5.	$\leq$	20.

Column Names / Row names

Example of Sample MPS Input

(further details are given in Chapter I of the documentation)

```
NAME    LP
ROWS
 N    OBJ
 L    RWN1
 L    RWN2

COLUMNS
      CLM1  OBJ  1.0  RWN1  2.0
      CLM2  OBJ  1.0  RWN2  4.0
      CLM3  OBJ  1.0  RWN1  3.0
      CLM3  RWN2 5.0

RHS
      RTH   RWN1  10.0
      RTH   RWN2  20.0

BOUNDS
 UP   BWN   CLM1  100.

ENDATA
```
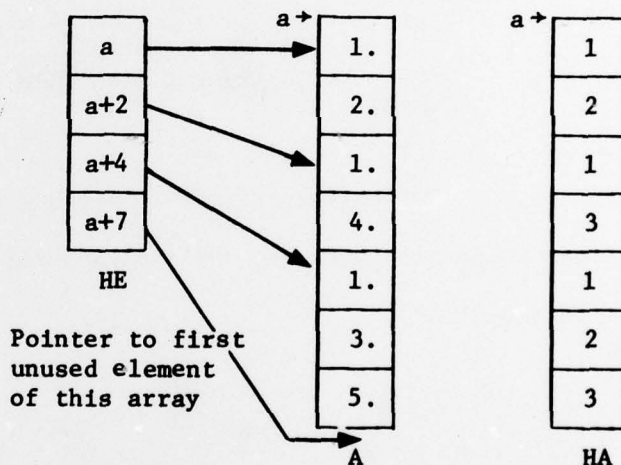
Packed Representation of Above Matrix, Excluding RHS (column list/row index data structure) (further details are given in Chapter I of the documentation)

Column Pointers: a, a+2, a+4, a+7, HE — Pointer to first unused element of this array

Matrix Elements (A): 1., 2., 1., 4., 1., 3., 5.

Row Indices (HA): 1, 2, 1, 3, 1, 2, 3

Thus the third column (called CLM3) starts at element a+4 of the array called 'matrix elements.' This column has three elements whose corresponding indices are given by the elements of the array called 'row indices.'

9

3.2.1. <u>Data Structure Manipulation Modules</u>

These carry out a number of operations on packed LP matrices. By using them it is possible to make a distinction between devising a strategy and the actual mechanics of implementing it.

The routines that are included in the current version of the collection are as follows:

- ADCONC -- concatenate two packed data structures and return result in first one
- ADRNDX -- reindex rows in a packed data structure
- ADINTF -- convert packed data structure to element/row index/column index data structure
- ADUPKC -- unpack a column of a packed data structure
- ADDELC -- delete a column in a packed data structure.

Obviously there are many other operations of this sort. The above are some which arise in the implementation of decomposition algorithms (c.f. Introduction).

3.2.2. <u>Simplex Modules</u>

Algorithms for structured LP are usually based upon repeated calls to a routine which performs one or more iterations of the revised simplex method. This routine must be specifically designed to meet the needs of the calling algorithm for structured LP, but the task of implementing it is made much easier if the following set of basic operations are available:

10

- MODRHS   given values of non-basics, develop modified right-hand side.
- FORMC   form cost vector for Phases 1 or 2 of Simplex method
- PRICE   price out columns
- CHVZR   determine which column leaves basis
- UPBETA   update current approximation to solution

All these modules utilize a common data structure, as shown in Fig. 6, and it is assumed that the packed LP matrix is in 'computational canonical form,' as explained in Fig. 6. In particular:

1. All bounds on the <u>structural</u> variables $\underline{x}$ are set in BL and BU. If a variable is unbounded above or below, the corresponding element of BL or BU is set to a machine representation of $\pm \infty$. (This is done by PREADB.)

2. A full identity matrix for the <u>logical</u> variables $(z_0, \underline{z})$ is assumed to be written at the start of the matrix. The bounds on these variables are determined by the type of rows (as specified in the rows section and read by PREADR). Again see Fig. 5. Thus no distinction is really made between positive and negative slacks and artificial variables. They simply have different bounds that they must satisfy.

An extension of the simplex method is inherent in the design of the modules and the associated data structure, in that non-basic variables can be fixed at values between their bounds, as specified by PEG. This is related to the superbasic variables of Murtagh and Saunders [11], but the latter are used in a more powerful way, since an optimization is carried out in the subspace they define. The use of pegged variables involves some straightforward extensions to PRICE, CHUZR and UPBETA (see documentation for more detail). PEG contains the current value of <u>every</u> variable in the problem, both logicals and structurals. Thus there is some redundancy of information stored; but

11

FIGURE 5: COMPUTATIONAL CANONICAL FORM

min $\quad \underline{c} \cdot \underline{x}$

s.t. $\quad A \cdot \underline{x} \begin{array}{c} \leq \\ = \\ \geq \end{array} \underline{b}$

and

$\quad \underline{\ell} \leq \underline{x} \leq \underline{u}$

Some elements of $\underline{\ell}$ and $\underline{u}$ may be machine representations of $+$ or $-\infty$.

$\Downarrow$

min $\quad \underline{c} \cdot \underline{x}$

s.t. $\quad I\underline{z} + A\underline{x} = \underline{b}$

$\quad \underline{\ell} \leq x \leq \underline{u}$

and $\quad 0 \leq z_i \leq \infty \quad$ if row i is $\leq$.(non-negative slack)

$\quad -\infty \leq z_i \leq \infty \quad$ if row i is $\geq$.(non-positive slack)

$\quad 0 \leq z_i \leq 0 \quad$ if row i is $=$.(artificial)

$\Downarrow$

min $\quad -z_0$

s.t. $\quad z_0 + \underline{c} \cdot \underline{x} = 0$

$\quad Iz + A\underline{x} = b$

$\quad \underline{\ell} \leq \underline{x} \leq \underline{u}$

$\quad -\infty \leq z_0 \leq \infty$

$\quad \underline{z}$ bounded as above

COMPUTATIONAL

CANONICAL

FORM

$\underline{x}$-structural variables

$(z_0, \underline{z})$-logical variables

We define $\bar{A} \stackrel{\Delta}{=} \left[I \mid \dfrac{c}{A} \right]$ an NROWS × NCOLS matrix.

12

FIGURE 6. DATA STRUCTURE FOR SIMPLEX MODULES

The LP matrix shown here is the one given in Fig. 3, transformed into 'computational canonical form' (Fig. 5), and then packed.

Packed matrix in computational canonical form, i.e. the matrix $\bar{A}$ of Fig. 5, stored in a column list/row index data structure (Fig. 3).

Status of columns:	KINBAS
Lower bounds:	BL
Upper bounds:	BU
Pegged variables:	PEG
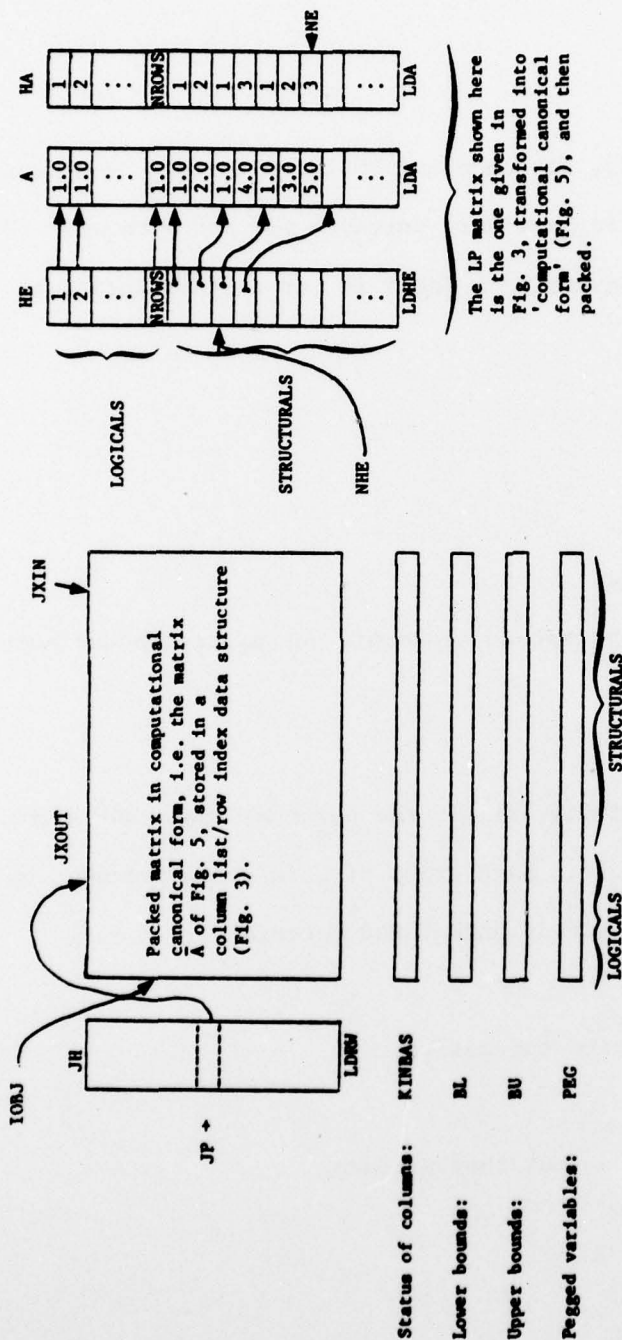
JH(I)   points to the I'th variable of the basis

KINBAS(J) = 0 if J'th variable is at lower bound
           = 1 if J'th variable is at upper bound
           = 2 if J'th variable is pegged between bounds
           = 3 if J'th variable is basic

JXIN   points to column to enter basis (determined by PRICE)

JP   points into JH and identifies which column JXOUT will exist from basis (determined by CHUZR)

A, HA, HE packed data structure. A and HA are of dimension LDA, and HE is of dimension LDHE

NE   number of elements in $\bar{A}$ (Fig. 5)

NHE   number of columns of $\bar{A}$.

13

this is not too great a penalty to pay in our experimental system, given the added flexibility that PEG makes possible.

2.2.2. Sparse Linear Algebra

The routines of Reid [12], implementing LU factorization of sparse matrices and Bartels-Golub updating are very suitable for our purposes, and we only provide an interface for converting a basis in our representation to one needed by Reid's routine LA05AD.

4. Documentation and Listings

In addition to this paper, machine-readable documentation has been written, organized as follows:

Chapter 1: User documentation giving briefly the purpose, usage and algorithmic details for each module. Each group of modules is preceded by an introductory section giving background material.

Chapter 2: Coding and documentation conventions.

Chapter 3: Testing programs and output they produce.

Chapter 4: Listings of modules.

5. Future Directions and Conclusions

One obvious direction for further development is the addition of new modules, for example, problem oriented modules to output solutions, more extensive data manipulation modules (e.g. to reorder a matrix by permutations), additional simplex modules (e.g. to deal with dual methods), additional sparse linear algebra routines, and so on.

A second direction for expansion is to make the modules easier to use, by deploying them within a suitable host system which takes over much of the bookkeeping tasks of a module and shields the user from them, e.g. the Speakeasy system of Cohen et al. [5]. In addition, the host system provides many additional facilities, e.g. graphical aids, extensive matrix manipulations, etc. In the calling sequence of a typical FORTRAN module, many parameters correspond to work vectors, work arrays, dimensioning information, switches and error flags. The Speakeasy language provides a very convenient mechanism for writing an interface to a FORTRAN subroutine. In essence the call can be redefined and often greatly simplified, since tasks of allocating work storage, parameter checking, etc., can be assumed by the interface. The MPL language [4] is another example of a suitable host environment.

The effort described in this paper is limited in scope and suffers from many shortcomings. However we would like, in conclusion, to emphasize three benefits that are gained from mathematical software patterned along these lines:

a) It is of use to investigators in Systems Optimization Laboratories who wish to implement optimization algorithms and study their behavior. In particular we plan to use our modules to implement algorithms for structured LP based upon the Dantzig-Wolfe Decomposition principle.
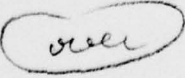
15

b) It is a useful educational aid; in particular our modules serve as a mini-
    tutorial on the implementation of LP algorithms, and we have already used
    them for this purpose in the lecture hall.

c) The discipline of systematizing and preparing codes raises many questions,
    which in turn spurs further research. The PEG array, see Fig. 6, is a small
    example of this. This point is also discussed, within the context of the
    LINPACK project, by Stewart [13, p. 7].

REFERENCES

[1] Nazareth, L. (1978). "Software For Optimization," Systems Optimization Laboratory Rept. SOL78-32, Department of Operations Research, Stanford University.

[2] Nazareth, L. (1977). "Minkit--an optimization system," Tech. Memo. 305, Applied Mathematics Division, Argonne National Laboratory.

[3] Boyle, J.M. and K.W. Dritz (1974) "An automated programming system to faciliate the development of quality mathematical software." Proceedings IFIP Congress '74, Stockholm, pp. 542-546.

[4] Dantzig, G.B. et al. (1970) "MPL-Mathematical Programming Language -- Specification Manual," Stanford University, Computer Science Department Rept. STAN-CS-70-187.

[5] Cohen, S. and S. C. Pieper (1976) "The SPEAKEASY-3 Reference Manual, Level Lamda", Argonne National Laboratory Rept. ANL-8000.

[6] Madsen, R.E. (1974) "Users Manual for SEXOP (Subroutines for Experimental Optimization), Release 4", Sloan School of Management, Massachusetts Institute of Technology.

[7] Land, A.H. and S. Powell (1973) Fortran Codes for Mathematical Programming, Wiley, London and New York.

[8] Dantzig, G.B. et al. (1973) "On the need for a Systems Optimization Laboratory," Optimization Methods for Resource Allocation, English University Press, London.

[9] Mathematical Programming System/360 Version 2, Linear and Separables Programming--User's Manual," IBM Document No. H20-0476-2, pp. 141-151.

[10] Saunders, M. A. (1977) "MINOS--Systems Manual," Systems Optimization Laboratory Rept SOL 77-31, Department of Operations Research, Stanford University.

[11] Murtagh, B. A. and M. A. Saunders (1977). "Nonlinear programming for large sparse systems," Systems Optimization Laboratory, Rept SOL 76-15, Department of Operations Research, Stanford University.

[12] Reid, J. K. (1976). "FORTRAN Subroutines for handling sparse linear programming bases," A.E.R.E, Harwell, Report R8269.

[13] Stewart G. W. (1977) "LINPACK Working Note 1," (manuscript).

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER SOL 78-28	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MODULES TO AID THE IMPLEMENTATION OF LP ALGORITHMS		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER SOL 78-28
7. AUTHOR(s) L. Nazareth		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0267
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Operations Research -- SOL Stanford University Stanford, CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR-047-143
11. CONTROLLING OFFICE NAME AND ADDRESS Operations Research Program -- ONR Department of the Navy 800 N. Quincy Street, Arlington, VA 22217		12. REPORT DATE December 1978
		13. NUMBER OF PAGES 17
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		16a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This report has been approved for public release and sale;
its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Large-Scale LP Software
Structured LP Problems
Modular Software for LP Problems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

SEE ATTACHED

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73
S/N 0102-014-6601

SOL 78-28
MODULES TO AID THE IMPLEMENTATION OF LP ALGORITHMS
L. Nazareth

Implementing large scale LP algorithms is a difficult, laborious and often
poorly rewarded task. This is particularly true of algorithms which exploit
the structure of the LP matrix. For this reason many algorithms have been
proposed in the literature, but few have been turned into good computer codes.
Very little is known about the relative performance of different algorithms.
In this paper we discuss some of the suggestions that have been made for
alleviating this problem and describe an approach based upon a carefully
defined collection of subroutines which are designed to aid the task of
implementing and comparing LP algorithms. These subroutines or modules may
be regarded as the 'primitives' of a language for implementing experimental
LP algorithms, particularly algorithms which exploit matrix structure. A
set of such modules is described. These have been implemented in FORTRAN,
and user documentation is available.